Cloning Strategies for MapReduce

Shahan Khatchadourian and Julia Rubin
Department of Computer Science
University of Toronto
{shahan, mjulia}@cs.toronto.edu

ABSTRACT
In this work, we discuss possible strategies for increasing the computational power of the MapReduce framework in a dynamic manner, using SnowFlock's cloning mechanism. We describe the implementation strategy that we have chosen and rationalize our decisions. We then discuss issues that we encountered during the implementation and the evaluation that we performed. Besides initial evaluation of the idea, our work provides ground and identifies issues related to the future exploration of this approach.

1. INTRODUCTION
MapReduce[1] is a programming model designed for processing large volumes of data in parallel by dividing the work into a set of independent tasks. It allows programmers to think in a data-centric fashion: they focus on applying transformations to sets of data records, and allow the details of distributed execution, network communication and fault tolerance to be handled by the MapReduce framework. The framework has become prevalent due to its simplicity as a cloud programming model.

Conceptually, a MapReduce program (also referred to as a *Job*) transform lists of input data elements into lists of output data elements. It does this in two phases: the *map phase* and the *reduce phase*. Each phase is made up of several *tasks* which run on a cluster of machines (*workers*).

Hadoop, a popular implementation of the MapReduce framework [2], is commonly installed on shared hardware controlled by virtual machine monitors (Cluster Setup Hadoop installation [3]). Such installations require identification and configuration of all machines in the cluster upfront. Adding a new machine to the cluster involves additional installation steps performed by a cloud administrator – a process that might take a significant amount of time ("minutes", according to [4]). Additionally, a job's configuration needs to be updated and may require the job itself to be restarted.

While Hadoop allows controlling the way cluster machines are used (by providing explicit configuration options that define the number of spawned map and reduce tasks for each job, as well as the number of map and reduce tasks for each worker), it does not provide a way to dynamically grow its computational power. Enhancing Hadoop with the ability to dynamically provision machines as a job is being processed is the main objective of our work. Towards this end, we propose to integrate Hadoop with SnowFlock [5] – a system that allows Xen virtual domains [6] to be *cloned* into impromptu clusters in a matter of sub-seconds. An application that has been designed to work in the SnowFlock environment should be able to expand its processing footprint in sub-second time, and then reduce it again when the computation is finished. Fast cloning is achieved by transmitting a VM state on demand, instead of replicating it upfront.

In what follows, we discuss the MapReduce framework in more details. We then explore different strategies for enhancing the Hadoop MapReduce implementation with the ability to expand its computational power in a dynamic manner. We discuss the strategies that we implemented for this project and evaluate our implementation by comparing the performance of the extended Hadoop system to the original one. Finally, we outline related approaches and discuss the future work.

2. MAP-REDUCE EXECUTION FLOW
A MapReduce program consists of two phases – map and reduce. A computation unit of each phase is referred to as a *task*. A special entity, called *master*, is responsible for keeping track of the job's execution and assigning tasks to workers.

The input to the map phase is a set of data files in an arbitrary format – line-based log files, multi-line input records, etc. These files are split into *input splits*, each of which describes a unit of work that comprises a single *map task* in a MapReduce program. The program executes as follows (see also Figure 1 for the high level architecture view):

1. A worker who is assigned a map task reads the contents of the corresponding input split. It parses key/value pairs out of the input data using the *record reader* and passes each pair to the *user-defined map function*. The intermediate key/value pairs produced by the map function are buffered in memory.
2. Periodically, the buffered pairs are written to local disk, partitioned into R regions, corresponding to R *reduce*

Produced on December 17, 2010
for CSC2231 Special Topics in Computer Systems: Cloud Computing.
Department of Computer Science, University of Toronto.

tasks. The locations of these buffered pairs on the local disk are passed back to the master, who is responsible for forwarding these locations to the reduce workers.

3. When a reduce worker is notified by the master about these locations, it uses remote procedure calls to read the buffered data from the local disks of the map workers (*shuffling*). When a reduce worker has read all of the intermediate data, it sorts it by the intermediate keys so that all occurrences of the same key are grouped together. The sorting is needed because typically many different keys map to the same reduce task.
4. The reduce worker iterates over the sorted intermediate data and for each unique intermediate key encountered, it passes the key and the corresponding set of intermediate values to *the user-defined reduce function*. The output of the reduce function is appended to a final output file for this reduce partition.

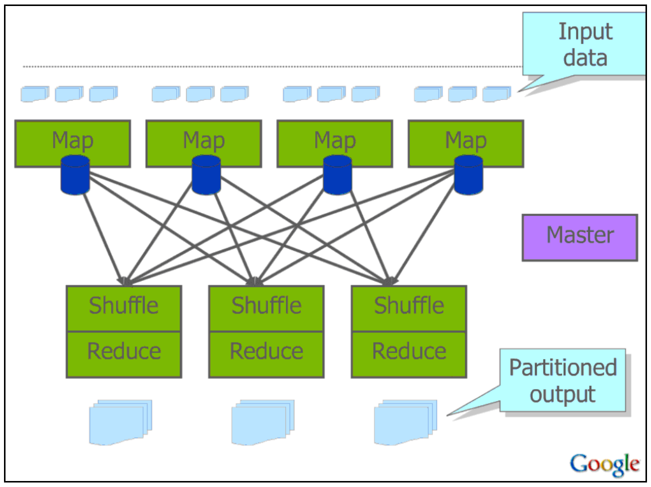

Figure 1 - MapReduce Framework.

Fault tolerance of the MapReduce job is ensured by the master, which pings every worker periodically and, if no response is received from a worker in a certain amount of time, marks the worker as failed. Any map task or reduce task in progress on a failed worker is reset to idle and becomes eligible for rescheduling.

Hadoop's MapReduce framework uses HDFS [7] as the underlying storage mechanism. HDFS supports fault-tolerance by replicating data to different nodes. Thus, intermediate results that are produced by the MapReduce jobs and stored in HDFS are protected by the same mechanism.

3. PROPOSED EXTENSIONS

We explored several strategies that extend Hadoop's MapReduce implementation with SnowFlock's ability to dynamically spawn new virtual machines. The most obvious approach involves introducing clones for increasing the number of map and reduce tasks that are able to execute in parallel. However, since the number of these tasks is defined by the number of input splits and the number of intermediate partitions, respectively, these numbers are known before the job begins executing and the desired multiplexing level can be achieved statically. On the other hand, the number of records processed by each map or reduce task is unknown in advance, because it depends on the input itself – the records in each input split and the number of keys in each reduce partition. Thus, we have chosen to focus on multiplexing the execution within a single map or reduce task. We define the following two cloning strategies:

1. Create clones as part of the map task, before the user-defined map functions are called; destroy them after they finish processing and returning their output to the map task.
2. Create clones as part of the reduce task, before the user-defined reduce functions are called; destroy them after they finish processing and returning their output to the reduce task.

However, due to the time constraints and due to the unexpected issues discussed in Section 5, we implemented cloning in the map phase only. We believe that the cloning in the reduce phase can be implemented in a similar manner.

In our architecture, the original map task creates clones to process tuples generated by the map's record reader. Each clone processes a subset of input tuples. As in the native MapReduce implementation, a tuple is processed by a call to the user-define map function. Tuples and their processing results (intermediate key/value pairs) are transferred between the original task and the spawned clones over the network. Intermediate results are then committed to disk by the original map task.

One consequence of this approach is the introduction of network overhead even for data-local map tasks.[1] However, allowing only the original map task to commit the produced intermediate key/value pairs to disk preserves MapReduce's fault-tolerance using Hadoop's native implementation: if some tuples are unprocessed because of clone or network failures, the map task is deemed incomplete and is rescheduled by the master. The simplicity of this fault-tolerance mechanism comes at a price: due to the increased number of clones, the potential for machine failure to disrupt the task increases as well, since if one clone fails, the entire task fails. We discuss possible solutions to this issue in Section 8.

[1] We investigate the overhead introduces by transferring tuples over the network in our evaluation.

4. PROTOTYPE IMPLEMENTATION

We implemented two versions of the prototype, varied by how clones are created and destroyed in the map phase.

In the first version, the map task fills *bucket*s of input tuples (configured to 1K tuples per bucket) and spawns a clone for each bucket once it finishes filling it. The clone is destroyed immediately after it finishes processing the bucket that was assigned to it. This method relies on having clones process buckets that are in memory due to the shared state that clones obtain from the original when they are spawned. It should be noted that besides the data associated with tuples, additional types of memory pages, such as kernel and user pages, are transferred by SnowFlock as well.

After implementing this version, it became apparent that it does not perform well due to the overhead of creating and destroying clones (and transferring the state for each created clone). Thus, we devised a second version that reuses clones. In this version, a fixed number of clones are created when a map task starts its execution. These clones live for the whole duration of the map task and are destroyed right before it finishes. They pull buckets of input tuples from the queue that the original map task populates, process them, and send the intermediate tuples back to the original.

An RPC protocol is used for communication purposes. Figure 2 outlines the details of the communication between an original and a cloned map tasks. An original map task, depicted on the left part of the figure, together with Hadoop's file system HDFS, reads the input data from HDFS locally. An RPC server, which is started by the original map task, is used to communicate with a clone, which is depicted on the right. RPC is used by clones to pull buckets, and then to send bucket of processed tuples back.

Cloning itself is conducted within a user-defined MapReduce job, thus, allowing all clone handling to be accomplished without modifying the Hadoop code. Embedding the cloning into the Hadoop infrastructure seems quite straightforward and is beneficial because this will allow running existing MapReduce jobs without modification. However, since the exact placement of the code that manages cloning does not influence the results of our evaluation, we have chosen the first approach and left embedding of the code into Hadoop framework for the future work.

5. DISCUSSION

While working on this project, we encountered several technical issues that prevented us to achieve the results that we expected. We discuss these issues below.

First, the stability of clones was unpredictable. Typically, after requesting an allocation of clones, clones would not start up at the same time, with delays ranging from several

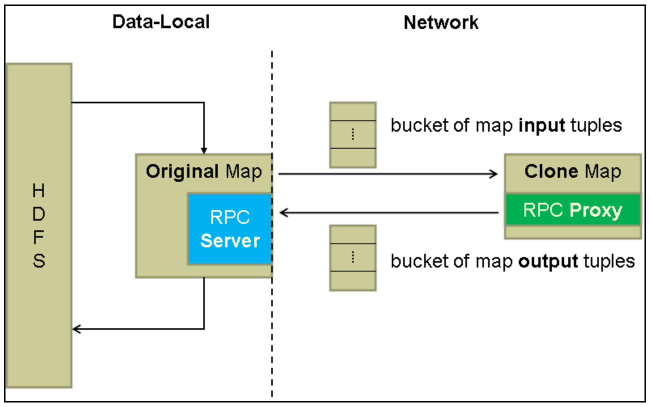

Figure 2 – Architecture.

seconds to minutes between clone starts. In some cases, several clones failed to start at all. In other cases, clones stopped responding for some time before coming back alive and continuing to process tuples. As a result, clones did not share the workload equivalently. Some discussions with the SnowFlock team members indicated this might be a network issue. It was beyond our ability to resolve this issue.

In addition, due to the memory limitations of the virtual machine, the number of tuples that the original map task could queue for the clones was severely limited, resulting in clones being idle. This affected the results of our experiments.

Due to the way originals and clones are networked, we were unable to perform evaluations when map tasks running in parallel on several VM spawn clones for the same job. This is because the network is setup so that original VMs can communicate with each other, as well as with their corresponding clones, but clones from one original cannot communicate with other originals and their clones. When clones are spawned from multiple machines in parallel, most were unable to communicate with the NameNode, which runs on one of the VMs. Deeper introspection of both SnowFlock and Xen scripts is required to implement our network requirements.

6. EVALUATION

We run experiments on a cluster of two machines with eight cores each. One machine was used as a SnowFlock master while the other was used for hosting clones. The master machine was preconfigured with a virtual machine running the Hadoop NameNode, JobTracker, TaskTracker and DataNode.

For our experiments, we used a bucket of 1K tuples. Some preliminary testing showed that smaller bucket sizes performed worse than larger bucket sizes. Figure 3 benchmarks the processing overhead as function of the bucket size – the number of tuples in a bucket that is passed from the original to the clone. The experiments were performed with one clone spawned from the original map task. While it can be seen that large buckets perform better than small buckets, we did not increase the bucket size beyond 1K in our experiment because the performance improvement was insignificant, while we had to avoid running out of memory, which happened due to the combination of two facts: (1) the available memory was limited and (2) clones sometimes became idle.

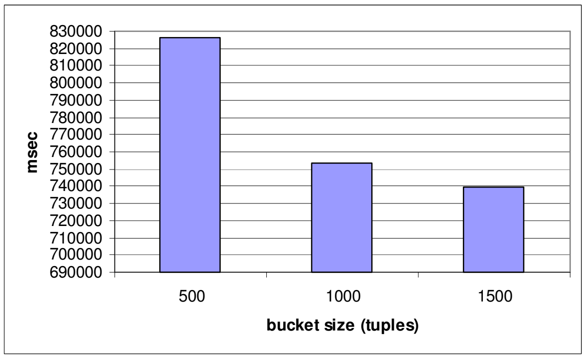

Figure 3 – Processing Overhead for Varying Bucket Size

To benchmark our implementation, we evaluated the performance of the WordCount MapReduce application on a 177M snapshot of the Wikipedia data available in an xml format. As the baseline, we measured the performance of the unmodified Hadoop processing the input in three different forms: (1) one virtual machine configured to run one map task at a time, (2) one machine configured to run three map tasks at a time, and (3) three machines configured to run one map task each (each machine performs a data-local map task). Since Hadoop uses 64M blocks, by default, the 177M input data is split into three input splits, which allowed us to initiate three concurrent map tasks, when needed.

We compared the performance of the unmodified Hadoop with the performance achieved with cloning. For cloning, we used one machine that runs one map task at a time and spawns various numbers of clones – from one clone per job up to six clones per job (the clone host we were provided with allows up to six clones). Also, as discussed in Section 5, we could not perform cloning from several machines in parallel due to the network configuration problems.

The results of our experiments are summarized in Figure 4. Each experiment was executed twice and we present an average between these two runs.[2] It is easy to see that the modified version (bars that correspond to 1-6 clones) performs worse that the original one. In addition, in spite of the expected result of having improved performance when increasing the number of clones, we observe a high variety in the results.

To further understand how the time is spent during a job's execution, we instrumented the code and performed micro-benchmarking, measuring time spent in specific activities during the execution. We benchmarked five such activities, presented in Figure 4:

1. *Clone create* is the time spent creating a specified number of clones.
2. *Clone destroy* is the time spent destroying the created clones.
3. *Map execution* is the time spent processing input tuples (i.e., time spent in the user-defined map function)
4. *Implementation and network overhead* is the time spent in our implementation, the major bulk of which involves the data transfer over the network.
5. *Hadoop processing* is the processing overhead introduced by the Hadoop framework itself.

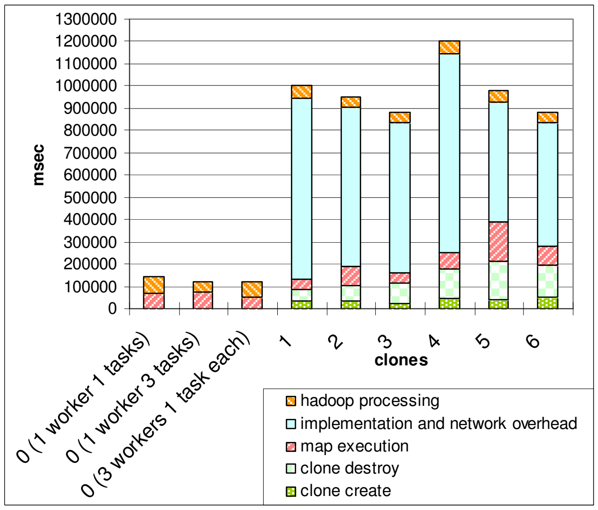

Figure 4 – Evaluation.

These detailed metrics revealed that the map execution time does not decrease linearly as the number of clones increases. We attribute that to the fact that clones sometimes become stale and/or network is unstable. We have also noticed that the time to create and, especially, to destroy clones took significantly longer than was indicated in [5]. Also, the higher is the number of created clones, the longer it takes to destroy them, while clone creation time does not display such behavior.

In addition, we observed that in our experiments, time spent in similar runs (e.g., same configuration, same number of clones) can vary significantly. We don't know the exact cause, for that but we noticed that generally rebooting the machines improve the stability and performance of subsequent runs. Another guess could be disk contention, or the network instability.

[2] Due to clone instability discussed in Section 5, we only have one run of the experiment with three and six clones.

Table 1 presents the detailed results from our experiments with cloning.

#	clone create	clone destroy	map execution	Impl. and network overhead	hadoop processing	total
1	31083	40715	63277	753683	50221	938979
1	34847	61026	37964	863104	73461	1044545
2	35405	69726	122478	739478	48209	1015296
2	34320	65425	46838	699349	45229	891161
3	20551	93633	46890	676826	42254	880154
4	42268	121146	62231	679973	44774	954311
4	44908	150166	84984	1110168	63857	1435406
5	47228	167336	269639	498080	48045	1030328
5	28670	183334	80255	585044	45990	923293
6	51968	145207	84725	552975	49615	884490

Table 1 – Detailed Results of the Experiments (msec).

As to our objective of multiplexing the map execution: even though we are able to observe some benefits from such multiplexing, e.g., map time for several experiments that use cloning are smaller than the map time of the original Hadoop with one map task[3] (due to the dynamic creation of new virtual machines and, thus, better load balancing), these benefits are not consistent and are lost altogether due to the implementation overhead that is introduced.

7. RELATED WORK

Enhancing an existing framework with SnowFlock's ability to dynamically clone virtual machines as required was instantiated in [8]. While in that work the authors combine SnowFlock with a parallel processing framework MPI, the idea behind the work is similar to ours – users of the system only need to maintain a fixed number of VMs, and install their usual applications. The system grows its computational power on demand and in a dynamic manner.

In MapReduce Online [9], the authors propose a modified MapReduce architecture in which intermediate data is pipelined between the map and the reduce phases, while preserving the programming interfaces and fault tolerance models of previous MapReduce frameworks. To implement pipelining, the data between the map and the reduce phases is passed over the network instead of being materialized onto the disk. We use similar technique for exchanging data between a map task and its spawned clones – map task sends to the clones input buckets over the network, and the clones send back the processed results.

8. FUTURE WORK

First, additional effort is required to make the implementation more robust to handle the issues outlined in Section 5.

[3] 69,74 msec comparing to the times shown in the forth column of Table 1.

Looking forward, the cloning strategy that we implemented supports coarse-grained fault recovery mechanism: if at least one of the spawned clones or the original fails, the whole task fails. As the number of clones per task increases, failures become more prevalent and they should be treated gracefully. Thus, a finer-grained fault recovery mechanism that tracks and re-execute only the tuples that were assigned to the failed clone could be implemented. Another possible extension is to track and re-execute slow "struggler" clones in order to improve the overall execution time of a task.

Instead of relying on the network to transfer data, a possibility of using a shared network-mounted disk that is mounted when a clone starts could be explored. However, this would incur a performance penalty due to the cost of serialization (in addition to the network overhead typically incurred for communicating with the shared disk).

Implementing cloning in the reduce phase, as well as implementing additional cloning strategies mentioned in Section 3 is another aspect of a possible future work.

9. ACKNOWLEDGMENTS

We wish to thank the SnowFlock team for their support during the implementation of this project.

10. REFERENCES

[1] Dean J, and Ghemawat S. MapReduce: Simplified data processing on large clusters. In *OSDI* (2004).

[2] Apache Hadoop MapReduce Project. http://hadoop.apache.org/mapreduce

[3] Apache Hadoop Cluster Setup. http://hadoop.apache.org/common/docs/current/cluster_setup.html

[4] Amazon Elastic Compute Cloud Developers Guide. http://docs.amazonwebservices.com/AWSEC2/latest/DeveloperGuide/

[5] Lagar-Cavilla H. A, Whitney J. A, Scannell A, Patchin P, Rumble S. M, De Lara E, Brudno M, and Satyanarayanan M. SnowFlock: Rapid Virtual Machine Cloning for Cloud Computing. In *Eurosys* 2009.

[6] Barham P, Dragovic B, Fraser K. Hand S, Harris T, Ho A, Neugebauer R, Pratt I, and Warfield, A. Xen and the art of virtualization. In *SOSP* 2003.

[7] Apache Hadoop Distributed File System http://hadoop.apache.org/hdfs/

[8] Patchin P, Lagar-Cavilla H.A, De Lara E, and Brudno M. Adding the Easy Button to the Cloud with SnowFlock and MPI. In *HPCVirt* (2009).

[9] Condie T, Conway N, Alvaro P, Hellerstein J.M, Elmeleegy K, and Sears, R. MapReduce Online. In *NSDI* (2010).